**NLP with Disaster Tweets**

**Author:** Maryam Moghadam

**Version:** 1

**Type of Investigation:** Predicting Disaster and non-disaster Tweets

**Goal of lab:** Twitter has become an important communication channel in times of emergency.  
The iniquitousness of smartphones enables people to announce an emergency they’re observing in real-time. Because of this, more agencies are interested in programmatically monitoring Twitter (i.e. disaster relief organizations and news agencies). In this lab you will build a machine learning model that predicts which Tweets are about real disasters and which one’s aren’t. You’ll have access to a dataset of 10,000 tweets that were hand classified.

**Operating Systems:** This project will be running on Code Ocean; you can run it either on the Text editor mode or Jupyter notebook. You can also implement this on a local Jupyter notebook.

**Hardware:** It’s recommended to use GPU to run the code faster (You may not be able to perform prediction of BERT on CPU)

**Support computer languages:** Python `

**Files/Data/Documents:** <https://www.kaggle.com/c/17777/download-all>

**Semester:** Spring 2020

**Date:** April 28, 2020

**Detail Procedure:**

**Part1: Downloading data**

1. Visit <https://www.kaggle.com/c/17777/download-all> and Download both the training and testing dataset and also the template for submission
2. Your training data includes 5 columns

|  |  |
| --- | --- |
| id | a unique identifier for each tweet |
| text | the text of the tweet |
| location | the location the tweet was sent from (may be blank) |
| keyword | a particular keyword from the tweet (may be blank) |
| target | in train.csv only, this denotes whether a tweet is about a real disaster (1) or not (0) |

1. You are predicting whether a given tweet is about a real disaster or not. If so, predict a 1. If not, predict a 0.

**Part2: Code Ocean environment setup**

1. In Code Ocean all projects must be done within capsules. You will be given a starter environment in which you can run your code from. You will be asked to sign up with your FAMU email.
2. Once Logged in click on new capsule and choose “create blank capsule” to create a new capsule and name it NLP
3. Now direct to your capsule and lets setup your environment
4. Once in the Capsule Click on “environment”
5. Now choose “Python with GPU support (3.7.3, miniconda 4.7.10)” environment

\*Note: This environment Includes CUDA 10.1 and cuDNN 7 support. conda makes this a great starting point for installing deep learning frameworks and other languages (including Python 2.7).

1. Now install all of the modules listed below using pip

* Bert-for-tf2
* Keras
* Nltk
* Numpy
* Pandas
* Pyspellchecker
* Scikit-learn
* Seaborn
* Sentencepiece
* Tenserflow
* Tensorflow-hub
* Tf-sentencepiece
* Tokenization
* Tqdm
* wordcloud

1. A screenshot of a cell phone

   Description automatically generatedYour environment should look like below
2. Now click on Jupyter notebook to launch a cloud workstation (Jupyter located on the right)

\* Note: a “localhost: \*\*\*\*/tree” will open in your currently browser

1. Now click on the Data folder and add the files that you have downloaded in Part1in this folder. Your data folder should include:

* train.csv - the training set
* test.csv - the test set
* sample\_submission.csv - a sample submission file in the correct format

1. Now navigate back and open the code folder and click on New then Python 3 to create a new python file name it whatever that is convenient for you
2. Now click on the created ipynb file and lets start Coding!

**Part3: Importing the libraries/modules and importing input data**

1. Let’s first import all the libraries and modules that are being used and is listed below

import os

import gc

import re

import string

import operator

import numpy as np # Linear algebra

import pandas as pd # Data processing, CSV file I/O (e.g. pd.read\_csv)

import seaborn as sns

import tensorflow as tf

import tensorflow\_hub as hub

from tensorflow import keras

import matplotlib.pyplot as plt

from wordcloud import STOPWORDS

from collections import defaultdict

from tensorflow.keras.optimizers import SGD, Adam

from tensorflow.keras.models import Model, Sequential

from sklearn.metrics import precision\_score, recall\_score, f1\_score

from sklearn.model\_selection import StratifiedKFold, StratifiedShuffleSplit

from tensorflow.keras.callbacks import ModelCheckpoint, EarlyStopping, Callback

from tensorflow.keras.layers import Dense, Input, Dropout, GlobalAveragePooling1D

from sklearn import feature\_extraction, linear\_model, model\_selection, preprocessing

from nltk.tokenize import word\_tokenize

from tqdm import tqdm

from nltk.corpus import stopwords

import re

import tensorflow as tf

from tensorflow import keras

from nltk.tokenize import word\_tokenize

from keras.preprocessing.text import Tokenizer

from keras.preprocessing.sequence import pad\_sequences

from keras.models import Sequential

from keras.layers import Embedding,LSTM,Dense,SpatialDropout1D

from keras.initializers import Constant

import nltk

nltk.download('stopwords')

nltk.download('punkt')

from sklearn.model\_selection import train\_test\_split

stop=set(stopwords.words('english'))

from tensorflow.keras.layer import Dense, Input

import tokenization

SEED = 1337

1. Now define your training and testing data frame and give it the path to the training and testing data files that you uploaded in your data folder

Example: # Reading the training and testing dataset

train\_df = pd.read\_csv("/root/capsule/data/train.csv")

test\_df = pd.read\_csv("/root/capsule/data/test.csv")

1. Now let’s take a look at both training and testing data frames to see how many columns and rows it has and how much memory it is taking, attach a screen shot of the result in the box below

A screenshot of a cell phone

Description automatically generated

**Part4: Analyzing Data**

Checking the class distribution of target value and also Character level, word level and sentence level analysis is very important in detection of disaster tweets since disaster tweets seem to be longer in length in comparison to non-disaster tweets. So, you should find Average word length in these tweets and understand the data you are dealing with and extract the meta data features of your dataset. Now let’s dive in and analyze our data

**A. Class distribution of target values:**

1. We will Extract the number of examples of each class of disaster and non-disaster tweets in the training data frame:

Example: Real\_len = train\_df[train\_df['target'] == 1].shape[0]

Not\_len = train\_df[train\_df['target'] == 0].shape[0]

1. Now let’s plot the two classes of disaster or non-disaster tweets using RcParams of Matplotlib and attach a screen shot of your plots in the box below:

A screenshot of a cell phone

Description automatically generated

1. You can also look for the missing values using the “isnull” in both training and testing datasets to just make sure your data frames distribution in all classes are close and totally different. This is imporant because you have to train and test your model with the same distribution to get accurate results.

**B. Finding length at (sentence, word, and character level):**

1. Now we will use the length function of python to check the length of all disaster and non-disaster tweets

Example: # This function returns the lenght of text

def length(text):

return len(text)

# Calculating lenght of text

train\_df['length'] = train\_df['text'].apply(length)

1. Now let’s plot the lenght of the text in Disaster and Non disaster tweets in the training data frame using RcParams of Matplotlib and attach a screen shot of your plots in the box below:

A close up of a map

Description automatically generated

1. As you can see the length of the non-disaster tweets are shorter that the disaster tweets
2. Now let’s look at the data at character level using the “st.len” function of python

Example: train\_len=train\_df[train\_df['target']==1]['text'].str.len()

train\_len=train\_df[train\_df['target']==0]['text'].str.len()

1. Now let’s use the map method to split the sentence of our training data frame and get their length

Example: train\_len=train\_df[train\_df['target']==1]['text'].str.split().map(lambda x: len(x))

train\_len=train\_df[train\_df['target']==0]['text'].str.split().map(lambda x: len(x))

1. At both step 7 and 9 you can plot these values to see the distribution on bar chart or any sort of diagram you like, this will help you to understand the data you are working with.
2. Finally let’s find the average length of words in tweets in the training data frame by using the apply method of python

Example:

word=train\_df[train\_df['target']==1]['text'].str.split().apply(lambda x : [len(i) for i in x])

word=train\_df[train\_df['target']==0]['text'].str.split().apply(lambda x : [len(i) for i in x])

1. Now take a plot of you result of the average word length in each tweet in the box below:

A screenshot of a cell phone

Description automatically generated

**Part5: Cleaning Data**

MetaData: Distributions of meta features in classes and datasets can help us to identify disaster tweets. Disaster tweets are usually written in a more formal way with longer words compared to non-disaster tweets (agencies). Non-disaster tweets have more typos than disaster tweets (individual users). The meta features used for the analysis in this project are:

|  |  |
| --- | --- |
| Word count | number of words in text |
| Unique word count | number of unique words in text |
| Stop word count | number of stop words in text |
| URL count | number of URLs in text |
| Punctuation count | number of punctuations in text |
| Hashtag count | number of hashtags (#) in text |
| Mean word length | average character count in words |
| Char count | number of characters in text |
| Mention count | number of mentions (@) in text |

1. Let’s start defining functions that can clean the meta features now using the available function of nltk library of python. Note: You have to define the removal function for all of the mentioned features
2. We should first use the Concatenate pandas objects along a particular axis with optional set logic along the other axes which in our case will be training and testing dataset then we will apply all the cleaining on the new pandas object that we get

Example: df=pd.concat([train\_df,test\_df])

1. Removing the URLs from the training and testing dataset

Example: def remove\_URL(text):

url = re.compile(r'https?://\S+|www\.\S+')

return url.sub(r'',text)

df['text']=df['text'].apply(lambda x : remove\_URL(x))

1. Removing HTML tags from the training and testing dataset:

Example: def remove\_html(text):

html=re.compile(r'<.\*?>')

return html.sub(r'',text)

df['text']=df['text'].apply(lambda x : remove\_html(x))

1. Removing the emojis from the traing and testing dataset

Example: def remove\_emoji(text):

emoji\_pattern = re.compile("["

u"\U0001F600-\U0001F64F" # emoticons

u"\U0001F300-\U0001F5FF" # symbols & pictographs

u"\U0001F680-\U0001F6FF" # transport & map symbols

u"\U0001F1E0-\U0001F1FF" # flags (iOS)

u"\U00002702-\U000027B0"

u"\U000024C2-\U0001F251"

"]+", flags=re.UNICODE)

return emoji\_pattern.sub(r'', text)

df['text']=df['text'].apply(lambda x: remove\_emoji(x))

1. Removing punstuations from training and testing dataset

Example: def remove\_punct(text):

table=str.maketrans('','',string.punctuation)

return text.translate(table)

df['text']=df['text'].apply(lambda x : remove\_punct(x))

1. Using PySpellChecker correct the spelling of training and testing dataset
2. Example: from spellchecker import SpellChecker

spell = SpellChecker()

def correct\_spellings(text):

corrected\_text = []

misspelled\_words = spell.unknown(text.split())

for word in text.split():

if word in misspelled\_words:

corrected\_text.append(spell.correction(word))

else:

corrected\_text.append(word)

return " ".join(corrected\_text)'''

df['text']=df['text'].apply(lambda x : correct\_spellings(x))

1. Now that you went through all the previous steps and cleaned your data, explain what are stopwords?

<https://towardsdatascience.com/treat-negation-stopwords-differently-according-to-your-nlp-task-e5a59ab7c91f>

**Part6: Data Embedding and Applying Machine Learning Algorithms**

**A. Baseline**

Scikit-learn's CountVectorizer counts the words in each tweet, turn them into data our model can process. Here is the list of the actions it does on our data:

|  |  |
| --- | --- |
| 1 | Tokenizing strings and giving an integer id for each possible token, for instance by using white-spaces and punctuation as token separators. |
| 2 | Counting the occurrences of tokens in each document. |
| 3 | Normalizing and weighting with diminishing importance tokens that occur in the majority of samples/documents |

Documents are described by word occurrences while completely ignoring the relative position information of the words in the document. If you want to read more about this process use the link below:

<https://scikit-learn.org/stable/modules/feature_extraction.html#text-feature-extraction>

As most documents will typically use a very small subset of the words used in the corpus, the resulting matrix will have many feature values that are zeros (typically more than 99% of them). For instance a collection of 10,000 short text documents (such as emails) will use a vocabulary with a size in the order of 100,000 unique words in total while each document will use 100 to 1000 unique words individually. In order to be able to store such a matrix in memory but also to speed up algebraic operations matrix / vector, implementations will typically use a sparse representation like implementations of the scipy.sparse package.

1. Now Let’s define a count vectorizer function:

Example: count\_vectorizer = feature\_extraction.text.CountVectorizer()

1. Now we will Creatw vectors for all of the tweets (both training and testing data frames)

Example: train\_vectors = count\_vectorizer.fit\_transform(train\_df["text"])

test\_vectors = count\_vectorizer.transform(test\_df["text"])

\*Note that we're NOT using .fit\_transform() here. Using just .transform() makes sure that the tokens in the train vectors are the only ones mapped to the test vectors - i.e. that the train and test vectors use the same set of tokens.

1. The length of the tweets and them being real disaster or not could be treated as a linear relation. These vectors are really big, to push model's weights toward 0 without completely discounting different ridge regression is used. Ridge regression is a way to create a parsimonious model when the number of predictor variables in a set exceeds the number of observations, or when a data set has correlations between predictor variables. If you would like to read more about it use this source: https://www.statisticshowto.com/ridge-regression/
2. Now we will define a classifier which is applying ridge regression

Example: clf = linear\_model.RidgeClassifier()

1. K-Fold Cross-validation is primarily used in applied machine learning to estimate the skill of a machine learning model on unseen data. That is, to use a limited sample in order to estimate how the model is expected to perform in general when used to make predictions on data not used during the training of the model. Portion of the known data is trained, then validated with the rest. If do this several times (with different portions) you will find how your model performed. The general procedure is as follows:

|  |  |
| --- | --- |
| 1 | Shuffle the dataset randomly. |
| 2 | Split the dataset into k groups |
| 3 | For each unique group:   |  |  | | --- | --- | | 3.1 | Take the group as a hold out or test data set | | 3.2 | Take the remaining groups as a training data set | | 3.3 | Fit a model on the training set and evaluate it on the test set | | 3.4 | Retain the evaluation score and discard the model | |
| 4 | Summarize the skill of the model using the sample of model evaluation scores |

You can read more about it from this link:

https://machinelearningmastery.com/k-fold-cross-validation/

In statistical analysis of binary classification, the F1 score is a measure of a test's accuracy.

sklearn.model\_selection.cross\_val\_score(estimator, X, y, cv, scoring=None)

estimator object implementing ‘fit’ The object to use to fit the data

The clf (for classifier) estimator instance is first fitted to the model; that is, it must learn from the model. This is done by passing our training set to the fit method.

X array-like: The data to fit. Can be for example a list, or an array

Y array-like: The target variable to try to predict in the case of supervised learning

scoring string: A string or a scorer callable object / function with signature scorer(estimator, X, y) which should return only a single value. Similar to cross\_validate but only a single metric is permitted. If None, the estimator’s default scorer (if available) is used.

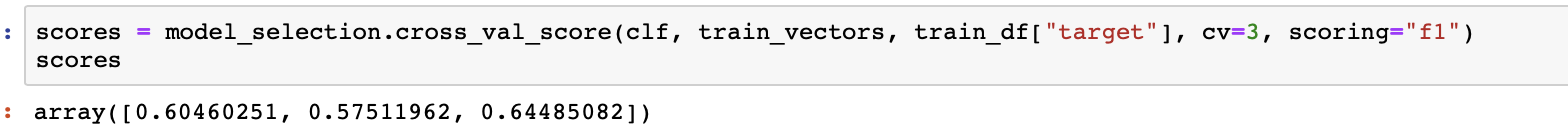
cv: int, Determines the cross-validation splitting strategy. integer, to specify the number of folds in a (Stratified)KFold

Return: scores: array of float, shape=(len(list(cv)),) Array of scores of the estimator for each run of the cross validation.

1. Now Lets applied what you just learnt and get the scorces:

Example: scores = model\_selection.cross\_val\_score(clf, train\_vectors, train\_df["target"], cv=3, scoring="f1")

1. Print the scores by adding “scores” in the next line of the command mentioned below and attach the results to the box below:



1. Next we will do predictions on training set and builde a submission. The clf (for classifier) estimator instance is first fitted to the model; that is, it must learn from the model. This is done by passing our training set to the fit method.

Example: clf.fit(train\_vectors, train\_df["target"])

1. Now you should read the template that you uploaded to the Data folder in Part1 and submit your results using the code below:

Example: sample\_submission = pd.read\_csv("/root/capsule/data/sample\_submission.csv")

sample\_submission["target"] = clf.predict(test\_vectors)

sample\_submission.to\_csv("submission-Baseline.csv", index=False)

1. Now if you go to the Code folder you can see that you have the result of your baseline models prediction added there. Print the first few of them and attach it in the box below:

A screenshot of a cell phone

Description automatically generated

**B. GLove**

GloVe, is an “unsupervised learning algorithm for obtaining vector representations for words.” Simply put, GloVe allows us to take a corpus of text, and intuitively transform each word in that corpus into a position in a high-dimensional space. This means that similar words will be placed together. It generate a single "word embedding" representation for each word in the vocabulary, so bank would have the same representation in bank deposit and river bankIn this section we will Use GloVe pretrained corpus model to represent our words.It is available in 4 varieties 25D, 50D ,100D and 200 Dimentional. We will try 100 D in this project. You can read more about it in this link: <https://nlp.stanford.edu/projects/glove/>.

1. Define a function that would let us to create a GLoVe corpus on our pandas dataframe

Example: def create\_corpus(df):

corpus=[]

for train\_df in tqdm(df['text']):

words=[word.lower() for word in word\_tokenize(train\_df) if((word.isalpha()==1) & (word not in stop))]

corpus.append(words)

return corpus

1. Now apply it on the data frame and crate a corpus

Example: corpus=create\_corpus(df)

1. Now download the pre trained Glove dictionary from the Stanford university and add it to you Data directory

<https://nlp.stanford.edu/projects/glove/>

1. Now add the pre trained Glove dictionary to your embediing dictionary

Example: embedding\_dict={}

with open('/root/capsule/data/Glove/glove.twitter.27B.100d.txt','r') as f:

for line in f:

values=line.split()

word=values[0]

vectors=np.asarray(values[1:],'float32')

embedding\_dict[word]=vectors

f.close()

1. We will then tokenize our data with GloVe’s tokenization index function and count the number of unique words

Example: MAX\_LEN=50

tokenizer\_obj=Tokenizer()

tokenizer\_obj.fit\_on\_texts(corpus)

sequences=tokenizer\_obj.texts\_to\_sequences(corpus)

train\_pad=pad\_sequences(sequences,maxlen=MAX\_LEN,truncating='post',padding='post')

1. Using the word index Report the number of unique words in the box below:

A screenshot of a cell phone

Description automatically generated

1. Now apply the embedding on the unique words

Example: num\_words=len(word\_index)+1

embedding\_matrix=np.zeros((num\_words,100))

for word,i in tqdm(word\_index.items()):

if i > num\_words:

continue

emb\_vec=embedding\_dict.get(word)

if emb\_vec is not None:

embedding\_matrix[i]=emb\_vec

1. We will now build our model with three layers. The first layer is the result of the embedding that we just got which will be our input layer, the second layer will be a dropout layer and the Third layer is a LSTM layer which is a Recurrent neural network architecuter of deep learning that is designed to save memory usage while training, and finally the last layerwhich is our output layer, is a dence layer with the sigmoid activation function.

Example: model=Sequential()

embedding=Embedding(num\_words,100,embeddings\_initializer=Constant(embedding\_matrix),

input\_length=MAX\_LEN,trainable=False)

model.add(embedding)

model.add(SpatialDropout1D(0.2))

model.add(LSTM(64, dropout=0.2, recurrent\_dropout=0.2))

model.add(Dense(1, activation='sigmoid'))

optimzer=Adam(learning\_rate=1e-5)

model.compile(loss='binary\_crossentropy',optimizer=optimzer,metrics=['accuracy'])

1. Build your model and attach a summary of your model in the box below:

A screenshot of a cell phone

Description automatically generated

1. Now lets apply the model on the training data frame with the batch size of 4 and 15 epochs and run the model.

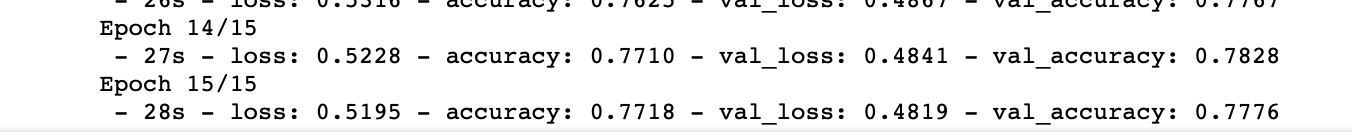
Examplele: train=train\_pad[:train\_df.shape[0]]

test=train\_pad[train\_df.shape[0]:]

X\_train,X\_test,y\_train,y\_test=train\_test\_split(train,train\_df['target'].values,test\_size=0.15)

history=model.fit(X\_train,y\_train,batch\_size=4,epochs=15,validation\_data=(X\_test,y\_test),verbose=2)

1. Now report the result of your training, accuracy and loss value of the last two epoch in the box below:



1. Now we will do prediction on ou testing data frame and submit our results as the submission-Glove

Example: y\_pre=model.predict(test)

y\_pre=np.round(y\_pre).astype(int).reshape(3263)

sub=pd.DataFrame({'id':submission['id'].values.tolist(),'target':y\_pre})

sub.to\_csv('submission-GLoVe.csv',index=False)

**C. Bert**

Bert is a Contextual model which generates a representation of each word that is based on the other words in the sentence. BERT makes use of Transformer, an attention mechanism that learns contextual relations between words (or sub-words) in a text. In its vanilla form, Transformer includes two separate mechanisms: an encoder that reads the text input and a decoder that produces a prediction for the task. Since BERT’s goal is to generate a language model, only the encoder mechanism is necessary. As opposed to directional models, which read the text input sequentially (left-to-right or right-to-left), the Transformer encoder reads the entire sequence of words at once. Therefore, it is considered bidirectional, though it would be more accurate to say that it’s non-directional. This characteristic allows the model to learn the context of a word based on all of its surroundings (left and right of the word).

1. Define the Bert encoding function and inside it define the tokens variable, masks variable and the segments variable, then in a for loop that is going through all of your text data tokenize your texts and append masks and segment to the generated vectors

\*Note that you have to return your vectors as a NumPy array

Example: def bert\_encode(texts, tokenizer, max\_len=512):

all\_tokens = []

all\_masks = []

all\_segments = []

for text in texts:

text = tokenizer.tokenize(text)

text = text[:max\_len-2]

input\_sequence = ["[CLS]"] + text + ["[SEP]"]

pad\_len = max\_len - len(input\_sequence)

tokens = tokenizer.convert\_tokens\_to\_ids(input\_sequence)

tokens += [0] \* pad\_len

pad\_masks = [1] \* len(input\_sequence) + [0] \* pad\_len

segment\_ids = [0] \* max\_len

all\_tokens.append(tokens)

all\_masks.append(pad\_masks)

all\_segments.append(segment\_ids)

return np.array(all\_tokens), np.array(all\_masks), np.array(all\_segments)

1. Now we will build our model
2. Define the function which is our model and name it Build model. This function will get the Bert layer as its input which will be downloaded from the tensorflow hub. Your output layer should be a dense layer with the sigmoid activation function:

Example:

def build\_model(bert\_layer, max\_len=512):

input\_word\_ids = Input(shape=(max\_len,), dtype=tf.int32, name="input\_word\_ids")

input\_mask = Input(shape=(max\_len,), dtype=tf.int32, name="input\_mask")

segment\_ids = Input(shape=(max\_len,), dtype=tf.int32, name="segment\_ids")

\_, sequence\_output = bert\_layer([input\_word\_ids, input\_mask, segment\_ids])

clf\_output = sequence\_output[:, 0, :]

out = Dense(1, activation='sigmoid')(clf\_output)

model = Model(inputs=[input\_word\_ids, input\_mask, segment\_ids], outputs=out)

model.compile(Adam(lr=2e-6), loss='binary\_crossentropy', metrics=['accuracy'])

return model

1. Now we will download the TensorFlow hubs Bert embedding from the website below: <https://tfhub.dev/tensorflow/bert_en_uncased_L-24_H-1024_A-16/1>

And we will add it to our Bert layer

Example:

module\_url = "https://tfhub.dev/tensorflow/bert\_en\_uncased\_L-24\_H-1024\_A-16/1"

bert\_layer = hub.KerasLayer(module\_url, trainable=True)

1. We will then define the vocabulary file that Bert will look into during the training from the vocabulary file assets of numpy

Example: vocab\_file = bert\_layer.resolved\_object.vocab\_file.asset\_path.numpy()

1. We will then define a function that would transfer our words into lowercase from numpy

Example: do\_lower\_case = bert\_layer.resolved\_object.do\_lower\_case.numpy()

1. Then we will define our tokenizer from the FullTokenizer of tensorflowhubs tokenization vector

Example: tokenizer = tokenization.FullTokenizer(vocab\_file, do\_lower\_case)

1. Now we have to apply the Bert encoder to our input layer of Bert for training data frame

Example: train\_input = bert\_encode(train\_df.text.values, tokenizer, max\_len=160)

1. Now we will apply the Bert encoder to our input layer of Bert for testing data frame

Example: test\_input = bert\_encode(test\_df.text.values, tokenizer, max\_len=160)

1. Now define your training labels that would be your target value

Example: train\_labels = train\_df.target.values

1. Now build the model

Example: model = build\_model(bert\_layer, max\_len=160)

1. Now take a summary of your model and attach a screen shot of it in the box below

A screenshot of a cell phone

Description automatically generated

1. And lets run it with three epochs and a batch size of 16

Example train\_history = model.fit(

train\_input, train\_labels,

validation\_split=0.2,

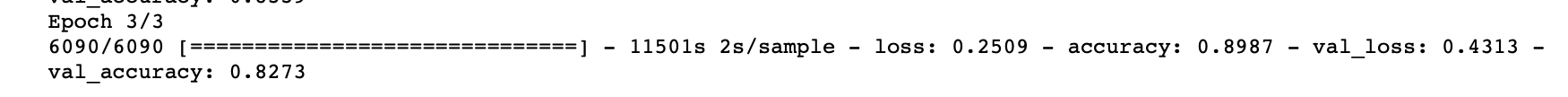
epochs=3,

batch\_size=16

)

model.save('model.h5')

1. Attach a screen shot of your last epochs result with the accuracy and loss value in the box:



1. Now we will do the prediction on our test data frame

Example: test\_pred = model.predict(test\_input)

1. And we will submit our results into a new CSV file name submission-Bert

Example: submission['target'] = test\_pred.round().astype(int)

submission.to\_csv('submission-Bert.csv', index=False)

1. Try adding more epoch to your training like 15 and see how your results changes and attach a screenshot of the new model’s last epochs accuracy value in the box below



**Critical Thinking:**

1. With which of the embedding algorithms did you get the best results?

Bert Algorithm

1. What is the main difference between Glove and Bert embeddings?

<https://www.quora.com/What-are-the-main-differences-between-the-word-embeddings-of-ELMo-BERT-Word2vec-and-GloVe>

Glove word embedding is context independent: Its output is just one vector (embedding) for each word, combining all the different sentences of the word into one vector (regardless of where the words occurs in a sentence and regardless of the different meanings they may have). If we had a sentence like “He went to the prison cell with his cell phone to extract blood cell samples from inmates”, where the word cell has different meanings based on the sentence context, these models just collapse them all into one vector for “cell” in their output.

BERT can generate different word embeddings for a word that captures the context of a word (that is its position in a sentence). For the same example of “He went to the prison cell with his cell phone to extract blood cell samples from inmates”, BERT would generate different vectors for the three vectors for cell. The first cell (prison cell case) , for instance would be closer to words like incarceration, crime etc. whereas the second “cell” (phone case) would be closer to words like iphone, android, galaxy

1. What are the other NLP tasks that these embeddings could be applied to?

<https://medium.com/future-vision/bert-meets-gpus-403d3fbed848>

question and answer, sentence classification, or sentiment analysis

1. How can you compare the Target column of your GLoVe prediction submission data frame and the BERT prediction submission Data frame using python coding and Pandas features to compare the result of your models?

<https://kanoki.org/2019/07/04/pandas-difference-between-two-dataframes/>

If we have to get all the rows which are not common between the two data frame or we want to see all the unique un-matched rows between two data frame then we can use the concat function with drop\_duplicate  
df1 = pd.read\_csv("/root/capsule/code/submission-Bert-15.csv")

df2 = pd.read\_csv("/root/capsule/code/submissionBert.csv")

pd.concat([df1,df2]).drop\_duplicates(keep=False)

**Final Report**

You can find the result for your project for your convenience to compare different filters that you implied. The source code for this project is also attached below.

<https://github.com/MaryamMoghadam/NLP-Disaster-Tweets/blob/master/Disaster_Detection.ipynb>

**Items Discussed:**

Code Ocean,Jupyter Notebook,Tensorflow, Tensorflow hub, matplotlib, keras, scikit learn, pandas, stopwords, tokenization, vectorization, concatenation, cross validation, ridg classifier, nltk, corpus, GloVe, BERT, drop duplicate

**Useful Resource:** You can find the ipynb file of this project on my Github repository

<https://github.com/MaryamMoghadam/NLP-Disaster-Tweets/blob/master/Disaster_Detection.ipynb>

<https://www.kaggle.com/c/nlp-getting-started/overview>